Занятие 8

«Пакет java.io и работа с ресурсами»

Состав

- Потоки данных и подклассы InputStream/OutputStream
- Сериализация java

План занятия

Потоки данных

InputStream и OutputStream

Базовый класс InputStream представляет классы, которые получают данные из различных источников:

- массив байтов
- строка (String)
- файл
- канал (pipe): данные помещаются с одного конца и извлекаются с другого
- последовательность различных потоков, которые можно объединить в одном потоке
- другие источники (например, подключение к интернету)

Для работы с указанными источниками используются подклассы базового класса InputStream.
Методы класса:

- int available() - возвращает количество байтов ввода, доступные в данный момент для чтения
- close() - закрывает источник ввода. Следующие попытки чтения передадут исключение IOException
- void mark(int readlimit) - помещает метку в текущую точку входного потока, которая остаётся корректной до тех пор, пока не будет прочитано readlimint байт
- boolean markSupported() - возвращает true, если методы mark() и reset() поддерживаются потоком
- int read() - возвращает целочисленное представление следующего доступного байта в потоке. При достижении конца файла возвращается значение -1
- int read(byte[] buffer) - пытается читать байты в буфер, возвращая количество прочитанных байтов. По достижении конца файла возвращает значение -1
- int read(byte[] buffer, int byteOffset, int byteCount) - пытается читать до byteCount байт в buffer, начиная с смещения byteOffset. По достижении конца файла возвращает -1
- reset() - сбрасывает входной указатель в ранее установленную метку
- long skip(long byteCount) - пропускает byteCount байт ввода, возвращая количество проигнорированных байтов

Класс OutputStream - это абстрактный класс, определяющий байтовый потоковый вывода.
Наследники данного класса определяют куда направлять данные: в массив байтов, в файл или канал. Из массива байт можно создать текстовую строку String.
Методы класса OutputStream :

- void write(int b) записывает один байт в выходной поток. Аргумент этого метода имеет тип int, что позволяет вызывать write, передавая ему выражение, при этом не нужно выполнять приведение его типа к byte.
- void write(byte b[]) записывает в выходной поток весь указанный массив байтов.

- void write(byte b[], int off, int len) записывает в поток часть массива len байтов, начиная с элемента b[off].
- void flush() очищает любые выходные буферы, завершая операцию вывода.
- void close() закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать IOException.

ByteArrayInputStream и ByteArrayOutputStream

Реализация стримов для чтения и записи в массив байт. Полезно для отладки.
Класс ByteArrayInputStream представляет входной поток, использующий в качестве источника данных массив байтов. Он имеет следующие конструкторы:

- ByteArrayInputStream(byte[] buf)
- ByteArrayInputStream(byte[] buf, int offset, int length)

В качестве параметров конструкторы используют массив байтов buf, из которого производится считывание, смещение относительно начала массива offset и количество считываемых символов length.
Считаем массив байтов и выведем его на экран:

```
import java.io.*;

public class FilesApp {

   public static void main(String[] args) {

      byte[] array1 = new byte[]{1, 3, 5, 7};
      ByteArrayInputStream byteStream1 = new ByteArrayInputStream(array1);
      int b;
      while((b=byteStream1.read())!=-1){

         System.out.println(b);
      }

      String text = "Hello world!";
      byte[] array2 = text.getBytes();
      ByteArrayInputStream byteStream2 = new ByteArrayInputStream(array2, 0, 5);
      int c;
      while((c=byteStream2.read())!=-1){

         System.out.println((char)c);
      }
   }
}
```

В отличие от других классов потоков для закрытия объекта ByteArrayInputStream не требуется вызывать метод close.

Класс ByteArrayOutputStream представляет поток вывода, использующий массив байтов в качестве места вывода. Чтобы создать объект данного класса, можно использовать один из его конструкторов :

```
ByteArrayOutputStream()
```

```
ByteArrayOutputStream(int size)
```

Первый конструктор создает массив данных для хранения байтов длиной в 32 байта, а второй конструктор создает массив длиной size.
Примеры использования класса ByteArrayOutputStream:

```
import java.io.ByteArrayOutputStream;

public class TestBOS
{
   public static void main(String[] args)
   {
      ByteArrayOutputStream bos = new ByteArrayOutputStream();
      String text = "Hello World!";
      byte[] buffer = text.getBytes();
      try{
         bos.write(buffer);
      } catch(Exception e) {
         System.out.println(e.getMessage());
      }
      // Преобразование массива байтов в строку
      System.out.println(bos.toString());

      // Вывод в консоль по символьно
      byte[] array = bos.toByteArray();
      for (byte b: array) {
         System.out.print((char)b);
      }
      System.out.println();
   }
}
```

В классе ByteArrayOutputStream метод write записывает в поток некоторые данные (массив байтов). Этот массив байтов записывается в объекте ByteArrayOutputStream в защищенное поле buf, которое представляет также массив байтов (protected byte[] buf). Так как метод write может вызвать исключение, то вызов этого метода помещается в блок try..catch.
Используя методы toString() и toByteArray(), можно получить массив байтов buf в виде текста или непосредственно в виде массива байт.
С помощью метода writeTo можно перенаправить массив байт в другой поток. Данный метод в качестве параметра принимает объект OutputStream, в который производится запись массива байт :
Для ByteArrayOutputStream не надо явным образом закрывать поток с помощью метода close.

FileInputStream и FileOutputStream

Реализация стримов для чтения и записи в файл.
Чтение файлов и класс FileInputStream

Для считывания данных из файла предназначен класс FileInputStream, который является наследником класса InputStream и поэтому реализует все его методы.
Для создания объекта FileInputStream мы можем использовать ряд конструкторов. Наиболее используемая версия конструктора в качестве параметра принимает путь к считываемому файлу:

- FileInputStream(String fileName) throws FileNotFoundException

Если файл не может быть открыт, например, по указанному пути такого файла не существует, то генерируется исключение FileNotFoundException.
Считаем данные из файла и выведем на консоль:

```
import java.io.*;

public class FilesApp {

   public static void main(String[] args) {

      try(FileInputStream fin=new FileInputStream("C://SomeDir//note.txt"))
      {
         System.out.println("Размер файла: " + fin.available() + " байт(а)");

         int i=-1;
         while((i=fin.read())!=-1){

            System.out.print((char)i);
         }
      }
      catch(IOException ex){

         System.out.println(ex.getMessage());
      }
   }
}
```

Подобным образом можно считать данные в массив байтов и затем производить с ним манипуляции:

```
byte[] buffer = new byte[fin.available()];
// считаем файл в буфер
fin.read(buffer, 0, fin.available());

System.out.println("Содержимое файла:");
for(int i=0; i<buffer.length;i++){

   System.out.print((char)buffer[i]);
}
```

Запись файлов и класс FileOutputStream

Класс FileOutputStream предназначен для записи байтов в файл. Он является производным от класса OutputStream, поэтому наследует всю его функциональность.
Например, запишем в файл строку:

```
import java.io.*;

public class FilesApp {

   public static void main(String[] args) {

      String text = "Hello world!"; // строка для записи
      try(FileOutputStream fos=new FileOutputStream("C://SomeDir//notes.txt"))
      {
         // перевод строки в байты
         byte[] buffer = text.getBytes();

         fos.write(buffer, 0, buffer.length);
      }
```

```
        catch(IOException ex){

            System.out.println(ex.getMessage());
        }
    }
}
```

Для создания объекта FileOutputStream используется конструктор, принимающий в качестве параметра путь к файлу для записи. Так как здесь записываем строку, то ее надо сначала перевести в массив байтов. И с помощью метода write строка записывается в файл.

Необязательно записывать весь массив байтов. Используя перегрузку метода write(), можно записать и одиночный байт:

```
fos.write(buffer[0]); // запись первого байта
```

Совместим оба класса и выполним чтение из одного и запись в другой файл:

```
import java.io.*;

public class FilesApp {

    public static void main(String[] args) {

        try(FileInputStream fin=new FileInputStream("C://SomeDir//notes.txt");
            FileOutputStream fos=new FileOutputStream("C://SomeDir//notes_new.txt"))
        {
            byte[] buffer = new byte[fin.available()];
            // считываем буфер
            fin.read(buffer, 0, buffer.length);
            // записываем из буфера в файл
            fos.write(buffer, 0, buffer.length);
        }
        catch(IOException ex){

            System.out.println(ex.getMessage());
        }
    }
}
```

Классы FileInputStream и FileOutputStream предназначены прежде всего для записи двоичных файлов. И хотя они также могут использоваться для работы с текстовыми файлами, но все же для этой задачи больше подходят другие классы.

BufferedInputStream и BufferedOutputStream

Декорированные типы для буферизации потока информации

Класс BufferedInputStream

Класс BufferedInputStream накапливает вводимые данные в специальном буфере без постоянного обращения к устройству ввода:

```
import java.io.*;

public class FilesApp {
```

```
    public static void main(String[] args) {

        String text = "Hello world!";
        byte[] buffer = text.getBytes();
       ByteArrayInputStream in = new ByteArrayInputStream(buffer);

       try(BufferedInputStream bis = new BufferedInputStream(in)){

          int c;
          while((c=bis.read())!=-1){

             System.out.print((char)c);
          }
       }
       catch(Exception e){

          System.out.println(e.getMessage());
       }
       System.out.println();
    }
}
```

Класс BufferedInputStream в конструкторе принимает объект InputStream. В данном случае таким объектом является экземпляр классаByteArrayInputStream.

Как и все потоки ввода BufferedInputStream обладает методом read(), который считывает данные. И здесь мы считываем с помощью метода read каждый байт из массива buffer.

Фактические все то же самое можно было сделать и с помощью одного ByteArrayInputStream, не прибегая к буферизированному потоку. Класс BufferedInputStream просто оптимизирует производительность при работе с потоком ByteArrayInputStream.

Класс BufferedOutputStream

Класс BufferedOutputStream аналогично создает буфер для потоков вывода. Этот буфер накапливает выводимые байты без постоянного обращения к устройству. И когда буфер заполнен, производится запись данных. Рассмотрим на примере:

```
import java.io.*;

public class FilesApp {

    public static void main(String[] args) {

       String text = "Hello world!"; // строка для записи
       try(FileOutputStream out=new FileOutputStream("notes.txt");
            BufferedOutputStream bos = new BufferedOutputStream(out))
       {
          // перевод строки в байты
          byte[] buffer = text.getBytes();
          bos.write(buffer, 0, buffer.length);
       }
       catch(IOException ex){
```

```
            System.out.println(ex.getMessage());
        }
    }
}
```

Класс BufferedOutputStream в конструкторе принимает в качестве параметра объект OutputStream - в данном случае это файловый поток вывода FileOutputStream. И также производится запись в файл. Опять же BufferedOutputStream не добавляет много новой функциональности, он просто оптимизирует действие потока вывода.

DataInputStream и DataOutputStream

Реализация стримов для работы с остальными примитивными типами и со String

Запись данных и DataOutputStream

Класс DataOutputStream представляет поток вывода и предназначен для записи данных примитивных типов, таких, как int, double и т.д. Для записи каждого из примитивных типов предназначен свой метод:

- writeBoolean(boolean v) : записывает в поток булевое однобайтовое значение
- writeByte(int v): записывает в поток 1 байт, которые представлен в виде целочисленного значения
- writeChar(int v): записывает 2-байтовое значение char
- writeDouble(double v): записывает в поток 8-байтовое значение double
- writeFloat(float v): записывает в поток 4-байтовое значение float
- writeInt(int v): записывает в поток целочисленное значение int
- writeLong(long v): записывает в поток значение long
- writeShort(int v): записывает в поток значение short
- writeUTF(String str): записывает в поток строку в кодировке UTF-8

Считывание данных и DataInputStream

Класс DataInputStream действует противоположным образом - он считывает из потока данные примитивных типов. Соответственно для каждого примитивного типа определен свой метод для считывания:

- boolean readBoolean(): считывает из потока булевое однобайтовое значение
- byte readByte(): считывает из потока 1 байт
- char readChar(): считывает из потока значение char
- double readDouble(): считывает из потока 8-байтовое значение double
- float readFloat(): считывает из потока 4-байтовое значение float
- int readInt(): считывает из потока целочисленное значение int
- long readLong(): считывает из потока значение long
- short readShort(): считывает значение short
- String readUTF(): считывает из потока строку в кодировке UTF-8
- int skipBytes(int n): пропускает при чтении из потока n байтов

Рассмотрим применение классов на примере:

```
import java.io.*;

public class FilesApp {

    public static void main(String[] args) {

        Person tom = new Person("Tom", 35, 1.75, true);
        // запись в файл
         try(DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.bin")))
```

```
        {
          // записываем значения
            dos.writeUTF(tom.name);
            dos.writeInt(tom.age);
            dos.writeDouble(tom.height);
            dos.writeBoolean(tom.married);
            System.out.println("Запись в файл произведена");
        }
        catch(IOException ex){

            System.out.println(ex.getMessage());
        }

        // обратное считывание из файла
        try(DataInputStream dos = new DataInputStream(new FileInputStream("data.bin")))
        {
          // записываем значения
            String name = dos.readUTF();
            int age = dos.readInt();
            double height = dos.readDouble();
            boolean married = dos.readBoolean();
            System.out.printf("Человека зовут: %s , его возраст: %d , его рост: %f метров, женат/замужем: %b",
                    name, age, height, married);
        }
        catch(IOException ex){

            System.out.println(ex.getMessage());
        }
    }
}

class Person
{
    public String name;
    public int age;
    public double height;
    public boolean married;

    public Person(String n, int a, double h, boolean m)
    {
        this.name=n;
        this.height=h;
        this.age=a;
        this.married=m;
    }
}
```

Здесь мы последовательно записываем в файл данные объекта Person.

Объект DataOutputStream в конструкторе принимает поток вывода: DataOutputStream (OutputStream out). В данном случае в качестве потока вывода используется

объект FileOutputStream, поэтому вывод будет происходить в файл. И с помощью выше рассмотренных методов типа writeUTF() производится запись значений в бинарный файл. Затем происходит чтение ранее записанных данных. Объект DataInputStream в конструкторе принимает поток для чтения: DataInputStream(InputStream in). Здесь таким потоком выступает объект FileInputStream

ObjectInputStream и ObjectOutputStream и сериализация

Реализация стримов для чтения/записи объектов с использованием механизма сериализации.
Сериализация представляет процесс записи состояния объекта в поток, соответственно процесс извлечения или восстановления состояния объекта из потока называется десериализацией. Сериализация очень удобна, когда идет работа со сложными объектами.

Интерфейс Serializable

Сразу надо сказать, что сериализовать можно только те объекты, которые реализуют интерфейс Serializable. Этот интерфейс не определяет никаких методов, просто он служит указателем системе, что объект, реализующий его, может быть сериализован.

Сериализация. Класс ObjectOutputStream

Для сериализации объектов в поток используется класс ObjectOutputStream. Он записывает данные в поток.
Для создания объекта ObjectOutputStream в конструктор передается поток, в который производится запись:

```
ObjectOutputStream(OutputStream out)
```

Для записи данных ObjectOutputStream использует ряд методов, среди которых можно выделить следующие:

- void close(): закрывает поток
- void flush(): очищает буфер и сбрасывает его содержимое в выходной поток
- void write(byte[] buf): записывает в поток массив байтов
- void write(int val): записывает в поток один младший байт из val
- void writeBoolean(boolean val): записывает в поток значение boolean
- void writeByte(int val): записывает в поток один младший байт из val
- void writeChar(int val): записывает в поток значение типа char, представленное целочисленным значением
- void writeDouble(double val): записывает в поток значение типа double
- void writeFloat(float val): записывает в поток значение типа float
- void writeInt(int val): записывает целочисленное значение int
- void writeLong(long val): записывает значение типа long
- void writeShort(int val): записывает значение типа short
- void writeUTF(String str): записывает в поток строку в кодировке UTF-8
- void writeObject(Object obj): записывает в поток отдельный объект

Эти методы охватывают весь спектр данных, которые можно сериализовать.
Например, сохраним в файл один объект класса Person:

```
import java.io.*;

public class FilesApp {

    public static void main(String[] args) {

        try(ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.dat")))
```

```
        {
            Person p = new Person("Джон", 33, 178, true);
            oos.writeObject(p);
        }
        catch(Exception ex){

            System.out.println(ex.getMessage());
        }
    }
}
class Person implements Serializable{

    public String name;
    public int age;
    public double height;
    public boolean married;

    Person(String n, int a, double h, boolean m){

        name=n;
        age=a;
        height=h;
        married=m;
    }
}
```

Десериализация. Класс ObjectInputStream

Класс ObjectInputStream отвечает за обратный процесс - чтение ранее сериализованных данных из потока. В конструкторе он принимает ссылку на поток ввода:

```
1    ObjectInputStream(InputStream in)
```

Функционал ObjectInputStream сосредоточен в методах, предназначенных для чтения различных типов данных. Рассмотрим основные методы этого класса:

- void close(): закрывает поток
- int skipBytes(int len): пропускает при чтении несколько байт, количество которых равно len
- int available(): возвращает количество байт, доступных для чтения
- int read(): считывает из потока один байт и возвращает его целочисленное представление
- boolean readBoolean(): считывает из потока одно значение boolean
- byte readByte(): считывает из потока один байт
- char readChar(): считывает из потока один символ char
- double readDouble(): считывает значение типа double
- float readFloat(): считывает из потока значение типа float
- int readInt(): считывает целочисленное значение int
- long readLong(): считывает значение типа long
- short readShort(): считывает значение типа short
- String readUTF(): считывает строку в кодировке UTF-8
- Object readObject(): считывает из потока объект

Например, извлечем выше сохраненный объект Person из файла:

```
import java.io.*;
```

```
public class FilesApp {

   public static void main(String[] args) {

      try(ObjectInputStream ois = new ObjectInputStream(new FileInputStream("person.dat")))
      {
         Person p=(Person)ois.readObject();
         System.out.printf("Имя: %s \t Возраст: %d \n", p.name, p.age);
      }
      catch(Exception ex){

         System.out.println(ex.getMessage());
      }

   }
}
```

Теперь совместим сохранение и восстановление из файла на примере списка объектов:

```
import java.io.*;
import java.util.ArrayList;

public class FilesApp {

   public static void main(String[] args) {

      String filename = "people.dat";
      // создадим список объектов, которые будем записывать
      ArrayList<Person> people = new ArrayList();
      people.add(new Person("Том", 30, 175, false));
      people.add(new Person("Джон", 33, 178, true));

      try(ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(filename)))
      {
         oos.writeObject(people);
         System.out.println("Запись произведена");
      }
      catch(Exception ex){

         System.out.println(ex.getMessage());
      }

      // десериализация в новый список
      ArrayList<Person> newPeople;
      try(ObjectInputStream ois = new ObjectInputStream(new FileInputStream(filename)))
      {
         newPeople=(ArrayList<Person>)ois.readObject();
      }
      catch(Exception ex){

         System.out.println(ex.getMessage());
```

```
        }

        for(Person p : newPeople)
            System.out.printf("Имя: %s \t Возраст: %d \n", p.name, p.age);
    }
}
```

Сериализация подробнее

Три способа сериализации

Serializable

Интерфейс-маркер, не имеет методов
Externalizable

Необходимо реализовать методы сериализации/десериализации - writeExternal(ObjectOutput) и readExternal(ObjectInput)
Переопределение стандартного способа

В классе необходимо объявить методы private void writeObject(java.io.ObjectOutputStream out) throws IOException;
private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;
Условия сериализации
При использовании Serializable, если родительский класс не сериализуем, у него должен быть конструктор без аргументов
При использовании Externalizable, у объекта должен быть конструктор без аргументов
Граф сериализации

До этого мы рассматривали объекты, которые имеют поля лишь примитивных типов. Если же сериализуемый объект ссылается на другие объекты, их также необходимо сохранить (записать в поток байт), а при десериализации – восстановить. Эти объекты, в свою очередь, также могут ссылаться на следующие объекты. При этом важно, что если несколько ссылок указывают на один и тот же объект, то этот объект должен быть сериализован лишь однажды, а при восстановлении все ссылки должны вновь указывать на него одного. Например, сериализуемыйобъект A ссылается на объекты B и C, каждый из которых, в свою очередь, ссылается на один и тот же объект D. После десериализациине должно возникать ситуации, когда B ссылается на D1, а C – на D2, где D1 и D2 – равные, но все же различные объекты.
Для организации такого процесса стандартный механизм сериализации строит граф, включающий в себя все участвующие объекты и ссылки между ними. Если очередная ссылка указывает на некоторый объект, сначала проверяется – нет ли такого объекта в графе. Если есть – объект второй раз не сериализуется. Если нет – новый объект добавляется в граф.
При построении графа может встретиться объект, порожденный от класса, не реализующего интерфейс Serializable. В этом случае сериализация прерывается, генерируется исключение java.io.NotSerializableException.
Рассмотрим пример:

```
import java.io.*;
class Point implements Serializable {
  double x;
  double y;
  public Point(double x, double y) {
    this.x = x;
```

```
    this.y = y;
  }
  public String toString() {
    return "("+x+","+y+") reference="+super.toString();
  }
}
class Line implements Serializable {
  Point point1;
  Point point2;
  int index;
  public Line() {
    System.out.println("Constructing empty line");
  }
  Line(Point p1, Point p2, int index) {
    System.out.println("Constructing line: " + index);
    this.point1 = p1;
    this.point2 = p2;
    this.index = index;
  }
  public int getIndex() { return index; }
  public void setIndex(int newIndex) { index = newIndex; }
  public void printInfo() {
    System.out.println("Line: " + index);
    System.out.println(" Object reference: " + super.toString());
    System.out.println(" from point "+point1);
    System.out.println(" to point "+point2);
  }
}
public class Main {
  public static void main(java.lang.String[] args) {
    Point p1 = new Point(1.0,1.0);
    Point p2 = new Point(2.0,2.0);
    Point p3 = new Point(3.0,3.0);
    Line line1 = new Line(p1,p2,1);
    Line line2 = new Line(p2,p3,2);
    System.out.println("line 1 = " + line1);
    System.out.println("line 2 = " + line2);
    String fileName = "d:\\file";
    try{
      // записываем объекты в файл
      FileOutputStream os = new FileOutputStream(fileName);
      ObjectOutputStream oos = new ObjectOutputStream(os);
      oos.writeObject(line1);
      oos.writeObject(line2);
      // меняем состояние line1 и записываем его еще раз
      line1.setIndex(3);
      //oos.reset();
      oos.writeObject(line1);
      // закрываем потоки
      // достаточно закрыть только поток-надстройку
      oos.close();
      // считываем объекты
      System.out.println("Read objects:");
      FileInputStream is = new FileInputStream(fileName);
      ObjectInputStream ois = new ObjectInputStream(is);
      for (int i=0; i<3; i++) { // Считываем 3 объекта
        Line line = (Line)ois.readObject();
        line.printInfo();
      } ois.close();
    } catch(ClassNotFoundException e) {
      e.printStackTrace();
    } catch(IOException e) {
      e.printStackTrace();
```

```
    }
  }
}
```

В этой программе работа идет с классом Line (линия), который имеет 2 поля типа Point (линия описывается двумя точками). Запускаемый класс Main создает два объекта класса Line, причем, одна из точек у них общая. Кроме этого, линия имеет номер (поле index). Созданные линии (номера 1 и 2) записываются в поток, после чего одна из них получает новый номер (3) и вновь сериализуется.

Выполнение этой программы приведет к выводу на экран примерно следующего:

```
Constructing line: 1
Constructing line: 2
line 1 = Line@7d39
line 2 = Line@4ec
Read objects:
Line: 1
  Object reference: Line@331e
  from point (1.0,1.0) reference=Point@36bb
  to point (2.0,2.0) reference=Point@386e
Line: 2
  Object reference: Line@6706
  from point (2.0,2.0) reference=Point@386e
  to point (3.0,3.0) reference=Point@68ae
Line: 1
  Object reference: Line@331e
  from point (1.0,1.0) reference=Point@36bb
  to point (2.0,2.0) reference=Point@386e
```

Из примера видно, что после восстановления у линий сохраняется общая точка, описываемая одним и тем же объектом (хеш-код 386e).

Третий записанный объект идентичен первому, причем, совпадают даже объектные ссылки. Несмотря на то, что при записи третьего объекта значение index было изменено на 3, в десериализованном объекте оно осталось равным 1. Так произошло потому, что объект, описывающий первую линию, уже был задействован в сериализации и, встретившись во второй раз, повторно записан не был.

Чтобы указать, что сеанс сериализации завершен, и получить возможность передавать измененные объекты, у ObjectOutputStream нужно вызвать метод reset(). В рассматриваемом примере для этого достаточно убрать комментарий в строке

```
//oos.reset();
```

Если теперь запустить программу, то можно увидеть, что третий объект получит номер 3.

```
Constructing line: 1
Constructing line: 2
line 1 = Line@ea2dfe
line 2 = Line@7182c1
Read objects:
Line: 1
  Object reference: Line@a981ca
  from point (1.0,1.0) reference=Point@1503a3
  to point (2.0,2.0) reference=Point@a1c887
Line: 2
  Object reference: Line@743399
  from point (2.0,2.0) reference=Point@a1c887
  to point (3.0,3.0) reference=Point@e7b241
Line: 3
  Object reference: Line@67d940
  from point (1.0,1.0) reference=Point@e83912
  to point (2.0,2.0) reference=Point@fae3c6
```

Однако это будет уже новый объект, ссылка на который отличается от первой считанной линии. Более того, обе точки будут также описываться новыми объектами. То есть в новом сеансе все объекты были записаны, а затем восстановлены заново.

Несериализуемые поля

При стандартной сериализации поля с модификаторами static и transient не сериализуются

serialVersionUID

Предположим, что вы добавили атрибут к объекту, скомпилировали его и распространили код на все машины кластера приложения. Объект хранится на компьютере с одной версией кода сериализации, но доступен для других машин, которые могут иметь разные версии кода. Когда эти машины пытались десериализовать объект, часто происходили неприятные вещи.
Метаданные Java-сериализации – информация, включенная в двоичный формат сериализации, – имеют сложную структуру и решают многие проблемы, досаждающие ранним разработчикам промежуточного ПО. Но всех проблем они тоже не решают.
При Java-сериализации используется свойство serialVersionUID, которое помогает справиться с разными версиями объектов в сценарии сериализации. Вам не нужно декларировать это свойство для своих объектов; по умолчанию платформы Java использует алгоритм, который вычисляет его значение на основе атрибутов класса, его имени и положения в локальном кластере. В большинстве случаев это работает отлично. Но при добавлении или удалении атрибутов это динамически сгенерированное значение изменится, и среда исполнения Java выдаст исключение InvalidClassException.
Чтобы избежать этого, нужно взять в привычку явное объявление serialVersionUID:

```
import java.io.Serializable;
public class Person implements Serializable {
        private static final long serialVersionUID = 20100515;
        // и т.д.
}
```

Я рекомендую использовать некую схему для номера версии serialVersionUID (в примере я использую текущую дату), причем нужно объявить его как private static final с типом long.
Вы можете спросить, когда изменять это свойство? Краткий ответ заключается в том, что его нужно изменять его всякий раз при внесении несовместимых изменений в класс, что обычно означает, что вы удалили атрибут. Если на компьютере имеется версия объекта, в которой атрибут удален, и этот объект переносится на машину с версией объекта, в которой этот атрибут ожидается, могут произойти неприятности.
В качестве золотого правила, при каждом добавлении или удалении элементов класса (то есть атрибутов и методов) нужно изменять его serialVersionUID. Лучше получить исключение InvalidClassException на другом конце "провода", чем приложение с ошибкой из-за несовместимых изменений класса.

Практическое задание

Реализовать программу со следующим функционалом:

1. Возможность ввода команд из консоли
2. Возможность считывания информации из файла (путь к файлу задается параметром команды) и построения на основе этой информации коллекции объектов. Пример команды – parse input.txt
3. Возможность сериализовать объекты из коллекции в выходной файл. Пример команды – serialize output.data
4. Возможность десериализовать объекты из файла и вывести их на печать в консоль. Пример команды – deserialize output.data

Пример файла input.txt в приложении 1

Материалы для подготовки

1. Курс на intuit
   http://www.intuit.ru/studies/courses/16/16/info
   лекция №15
2. Сериализация объектов
   http://www.skipy.ru/technics/serialization.html

Вопросы для самоконтроля

1. Перечислите основные методы класс InputStream?
2. Что такое граф сериализации?
3. Какой паттерн проектирования использует java.io?
4. В чем назначение serialVersionUID?

Приложение 1

Пример содержимого файла input.txt (каждая строка содержит описание объекта Студент)

```
Name=Ivan age=20
Name=John age=30
Name=Kate age=25
Name=Kelly age=40
Name=Sergey age=35
```